Universidad de Lleida

Escuela Politécnica Superior

Ingeniería en Informática

Trabajo Final de Carrera

# Paralelización del algoritmo Rho de Pollard utilizando el protocolo de paso por mensajes MPI

Autor: Teodoro Andrés Laírla Morlans

Director: Francesc Sebé i Feixas

Septiembre de 2010

# Paralelización del Algoritmo Rho de Pollard utilizando el protocolo de paso por mensajes MPI

Teodoro Andrés Laírla Morlans

Septiembre de 2010

Quiero agradecer este proyecto al Departamento de Matemática, en especial a Francesc, Josep Maria, Núria, Ivan y Víctor, por la ayuda que me han brindado; y, por encima de todo, a mis padres, sin los que nada de esto hubiera sido posible.

# Índice general

# Capítulo 1

# Introducción

La proliferación de los equipos conectados a Internet desde finales de los 90 hasta hoy, ha provisto de conexión a la red a cualquier persona dentro del mundo desarrollado. Esto ha creado una economía y una base de negocio en la red, así como una manera fácil y cómoda de acceder a todo tipo de servicios; servicios que, en aras de preservar la privacidad de las personas que acceden a ellos, han tenido que blindarse con el fin de evitar ataques o filtraciones de información, no deseados por ninguna de las partes.

Se pone de manifiesto, pues, la necesidad de aplicar la criptología a los sistemas informáticos dando paso a todo un campo de investigación critográfica que, aprovechando todos los recursos y avances que ofrecen las computadoras, se desarrolla intentando ir un paso por delante de la vertiginosa manera de avanzar que tiene Internet.

La manera en que las computadoras aumentan sus capacidades, hace necesarios nuevos sistemas de cifrado mediante problemas matemáticos inabordables según el paradigma procedural existente. Paradigma al que, por otra parte, puede acceder cualquiera de nosotros que se pueda permitir las máquinas multiprocesador actualmente en mercado a precios realmente asequibles.

Para que un sistema criptográfico sea fiable, se tiene que estudiar, y se tienen que dedicar más horas a estudiar y probar la manera de descifrarlo, de las que se dedicaron para desarrollarlo. Esto es, el criptoanálisis, y por estos fueros anda este proyecto en el que se pone a prueba el problema del logaritmo discreto rediseñando el algoritmo Rho de Pollard para ejecutarlo en un entorno MPI. Se combinan así, el rendimiento en paralelo proporcionado por el algoritmo entrelazado con el entorno MPI, y el coste negligible en memoria de la adaptación del Rho de Pollard.

Para conseguir el objetivo propuesto, se han llevado a cabo una serie de pasos diferenciados necesarios que han permitido obtener resultados y extraer las conclusiones pertinentes; tanto de los resultados en sí, como del proceso seguido hasta obtenerlos. Entre esas fases en la realización del proyecto se pueden destacar:

- Estudio del algoritmo Rho de Pollard original.
- Estudio de sus diferentes paralelizaciones.
- Diseño de las estructuras que se utilizarían para implementar el algoritmo.
- Implementación de las estructuras y del algoritmo paralelo.
- Experimentación en el clúster.
- Análisis completo de los resultados.

# Capítulo 2

# Conceptos Previos

Para la correcta comprensión de este proyecto, se requieren, por parte del lector, una serie de conceptos matemáticos y criptográficos previos. En este apartado se explican la mayoría de estos conceptos asumiendo un conocimiento matemático previo y sin llegar a profundizar en ellos.

## 2.1. Conceptos Matemáticos

- **Conjunto:** Agrupación de objetos, llamados elementos, que se incluyen en el conjunto debido a una relación de pertenencia sabida. Lo que es lo mismo, dados un conjunto y un elemento siempre se puede discernir si el elemento pertenece al conjunto o no.
- **Cardinal de un conjunto:** Es el número de elementos que tiene un conjunto $G$. Se expresa como $|G|$, $\#G$ o $Card(G)$
- **Grupo:** Un conjunto G con una operación binaria $*$, se dice que es grupo si se cumplen las siguientes propiedades:

  1. **Operación interna o cerrada:** para dos elementos cualesquiera $x$ e $y$ del conjunto $G$ operados con la operación $*$, el resultado siempre pertenece al mismo conjunto $G$:

     $$\forall x, y \in G : x * y \in G$$

  2. **Asociatividad:** para cualesquiera elementos del conjunto $G$, no importa el orden en que se operen las parejas de elementos:

     $$\forall x, y \in G : x * (y * z) = (x * y) * z$$

  3. **Elementro neutro:** para todo elemento de $G$ existe un elemento único del conjunto con el que operado según la operación $*$, el resultado es el mismo elemento:

     $$\exists! e \in G \text{ tal que } \forall x \in G : e * x = x * e = x$$

4. **Elemento simétrico:** para todo elemento de $G$ existe otro elemento de $G$ tal que al operarse los dos elementos según la operación $*$ el resultado es el elemento neutro $e$:

$$\forall x \in G, \exists \bar{x} \in G \text{ tal que } \bar{x} * x = x * \bar{x} = e$$

- **Grupo Abeliano o Grupo Conmutativo:** Un grupo abeliano o grupo conmutativo es un grupo $(G, *)$ en el que su operación binaria satisface la propiedad conmutativa:

$$\forall x, y \in G : x * y = y * x$$

- **Orden de un Grupo:** El orden de un grupo $(G, *)$ es el cardinal del conjunto $G$.

- **Grupo Finito:** Se dice que un grupo $(G, *)$ es finito si el cardinal de $G$ es finito.

- **Orden de un Elemento:** El orden $n$ de un elemento $a$ de un grupo $(G, *)$ es el menor de los números naturales que cumple que al elevar el elemento a la potencia del número el resultado es el elementro neutro $e$.

$$a^n = a* \overbrace{\cdots}^{n} *a = e$$

- **Grupo Cíclico:** Un grupo cíclico $G$ es un grupo que puede ser generado por un solo elemento. Es decir, existe un elemento $g$, denominado elemento generador, tal que cualquier elemento de $G$ puede expresarse como una potencia de $g$. Por tanto un grupo $(G, *)$ es cíclico si $G = \{g^n | n \in \mathbb{Z}\}$

## 2.2. Conceptos Criptográficos

- **Criptología:** Ciencia que se basa en el estudio y/o arte de ocultar la información, y cuyas principales áreas de estudio son la criptografía y el criptoanálisis; pero que también incluye la estaganografía.

- **Criptografía:** Parte de la criptología que se basa en buscar sistemas de cifrado y descifrado de la información mediante claves.

- **Criptograma:** Mensaje cifrado cuyo significado resulta ininteligible hasta que es descifrado con la clave adecuada.

- **Criptoanálisis:** Parte de la criptología que se basa en el estudio de los métodos para obtener el mensaje original partidendo de criptogramas, sin tener acceso a ninguna de las claves secretas.

- **Criptosistema:** Un criptosistema o sistema criptográfico es una colección de transformaciones de texto en claro a texto cifrado y viceversa, en la que la transformación o transformaciones que se han de utilizar son seleccionadas por claves y están definidas normalmente por un algoritmo matemático.

## 2.3. Tipos de Criptosistemas

Existen dos tipos principales de criptosistemas, simétricos y asimétricos. La diferencia radicai en que en los simétricos, también llamados de clave compartida, se usa la misma clave para cifrar que para descifrar, aunque el procedimiento sea diferente; mientras que en los asimétricos, o de clave pública, las claves que se usan para cifrar o descifrar un mismo mensaje son diferentes, aunque guardan una relación matemática entre ellas.

### 2.3.1. Criptosistemas de Clave Compartida

Los criptosistemas de clave compartida o criptosistemas simétricos son criptosistemas en los que tanto el receptor como el emisor conocen una clave única llamada clave compartida, que le sirve al emisor para cifrar el mensaje antes de enviarlo y al receptor para descifrar el criptograma recibido del emisor. Son criptosistemas con los que se consigue una seguridad alta sin necesidad de utilizar un gran número de bits para el tamaño de la clave. Una vez emisor y receptor conocen la clave pueden enviar la información cifrada con seguridad. Sin embargo, el problema se plantea durante el intercambio de claves, ya que es difícil encontrar un canal seguro en el que acordar la clave privada.

Ejemplos de criptosistemas simétricos son DES, RC5, AES, Blowfish e IDEA, entre otros.

### 2.3.2. Criptosistemas de Clave Pública

Los criptosistemas de clave pública, o criptosistemas asimétricos, son métodos criptográficos que usan un par de claves relacionadas entre sí para el cifrado y descifrado de mensajes. Cada participante de la comunicación, estos son emisor y receptor, tienen un par de claves asociadas: una pública y otra privada.

La clave pública del receptor y, por ser pública, conocida por el emisor, la utiliza este último para cifrar el mensaje antes de enviarlo; mientras que la clave privada del receptor y, por ser privada, sólo conocida por el receptor mismo, se utiliza para descifrar el criptograma recibido cifrado con su clave pública.

Este modelo de criptosistemas soluciona el problema del intercambio de claves, ya que no importa que el canal no sea seguro para transmitir la clave pública, aunque es susceptible de sufir el ataque del intermediario (man in the middle).

El ataque del itermediario se produce cuando hay un atacante conectado a tu red en un canal no seguro. Si el atacante está escuchando el tráfico entre emisor y receptor, no le sería difícil interceptar el mensaje del emisor pidiendo la clave pública al receptor y enviar una clave pública generada por él, haciéndose pasar por el receptor. Si el intermediario es lo suficiemtemente hábil, conseguirá engañar al emisor para que le envíe el mensaje cifrado con la clave pública falsa y descifrarlo con su clave privada pareja de la falsa clave previamente enviada. Esto se soluciona con las entidades certificadoras y los certificados digitales.

Otras desventajas que tienen los criptosistemas de clave pública son: el tamaño de las claves, que es mayor al de las claves de los criptosistemas simétricos para niveles equivalentes de seguridad; el mayor coste computacional; y la limitación de tamaño de los criptogramas según sea el tamaño de la clave.

Algunos ejemplos de criptosistemas asimétricos son: RSA, ElGamal o la cifra de Paillier.

### 2.3.3. Criptosistemas híbridos

Los criptosistemas híbridos son criptosistemas internamente compuestos por dos criptosistemas, uno de clave pública y uno de clave privada. La idea de estos criptosistemas, es intercambiar la clave secreta del criptosistema simétrico mediante el criptosistema de clave pública. Así, no hace falta establecer un canal seguro para intercambiar la clave secreta, y se aprovechan las bondades de los sistemas simétricos, evitando el problema del intercambio de claves.

## 2.4. El problema del logaritmo discreto

Sea $G$ un grupo multiplicativo cíclico de orden primo $p$ y sea $g$ un generador de $G$. Dado $h \in G$, el problema del logaritmo discreto consiste en encontrar un $m$, tal que $g^m = h$. Lo que se puede reescribir como $m = \log_g h$. Cuando el orden del grupo $G$ tiene un factor primo grande, existen grupos donde el mejor algoritmo conocido para su resolución tiene coste no polinomial, haciendo a estos grupos adecuados para su uso en criptografía.

Cuando G es un subgrupo multiplicativo de un cuerpo de Galois $\mathbb{F}_q^*$, el algoritmo *index-calculus* calcula un logaritmo discreto con un coste subexponencial $O(e^{\sqrt{\log q \log\log q}})$. Si $q$ es primo, el algoritmo *number field sieve* reduce este coste a $O(e^{1{,}923(\log q)^{1/3}(\log\log q)^{2/3}})$.

Para otros grupos como el grupo de puntos de una curva elíptica o la variedad jacobiana de una curva hiperelíptica, el mejor algoritmo conocido es el algoritmo *Rho de Pollard*, cuyo coste (exponencial con la longitud de $p$) es $O(\sqrt{p})$. Debido a su coste exponencial, sólo se puede considerar el tiempo de ejecución de este algoritmo aceptable cuando $p$ tiene una longitud reducida. Para resolver instancias con longitudes de $p$ mayores será necesaria la utilización de varios procesadores ejecutando una versión paralelizada del algoritmo.

### 2.4.1. Criptosistema ElGamal

El criptosistema ElGamal es un criptosistema de clave pública que basa su seguridad en la intratabilidad del problema del logaritmo discreto.

#### Configuración

1. Se escogen dos números primos $p$ y $q$ tales que $p = 2q+1$. De esta manera, todos los elementos de $\mathbb{Z}_p^*$ tendrán orden 2, $q$ o $2q$.

2. Se elige un elemento $g$ de $\mathbb{Z}_p^*$ con orden $q$.
3. Se publican $p$ y $g$.

### Generación de claves

Partiendo de $p$ y $g$ y sabiendo que $q = (p-1)/2$

1. Clave privada $x$: Se escoge un número aleatorio $x$ tal que $0 < x < q$
2. Clave pública $y$: Se calcula $y = g^x \pmod{p}$

### Cifrado

- Entrada
  - Parámetro $p$.
  - Parámetro $g$.
  - Clave pública del receptor $y$.
  - Mensaje $M$ tal que $0 < M < p$
- Procedimiento
  - Se genera un número aleatorio $r$ tal que $0 < r < q$
  - Se calcula $C_1 = g^r$
  - Se calcula $C_2 = M \cdot y^r$
- Salida
  - Tupla $C = (C_1, C_2)$

Como la clave pública $y$ es conocida para todo el mundo, al igual que $g$; la dificultad de obtener la clave privada a partir de estos valores se basa en la dificultad de resolver el cálculo $x = \log_g y$. Esto es, precisamente, el problema del logaritmo discreto.

### Descifrado

- Entrada
  - Parámetro $p$.
  - Parámetro $g$.
  - Clave privada del receptor $x$.
  - Criptograma $C = (C_1, C_2)$
- Procedimiento
  - Se calcula $C_2 \cdot ({C_1}^x)^{-1} \pmod{p} = M \cdot y^r \cdot (g^{xr})^{-1} = M \cdot y^r \cdot (y^r)^{-1} = M$
- Salida
  - Mensaje en claro $M$

# Capítulo 3

# Algoritmo Rho de Pollard

El algoritmo Rho de Pollard resuelve el problema del logaritmo discreto sobre un grupo $G$ de orden $p$ (dados $g$, $h \in G$, calcula $\log_g h$) a partir de dos pares distintos $(a_i, b_i)$, $(a_j, b_j)$ de enteros módulo $p$ que satisfacen:

$$g^{a_i} h^{b_i} = g^{a_j} h^{b_j}$$

Operando la expresión anterior se obtiene,

$$h^{b_i - b_j} = g^{a_j - a_i}$$

lo que permite calcular $\log_g h$ como,

$$\log_g h = (a_j - a_i)(b_i - b_j)^{-1} \pmod{p}. \tag{3.1}$$

El algoritmo Rho de Pollard utiliza una función $f : G \to G$ tal que dada una tripleta $(x_i, a_i, b_i)$ que satisface $x_i = g^{a_i} h^{b_i} \in G$, es fácil calcular otra tripleta distinta $(x_j, a_j, b_j)$ tal que $x_j = f(x_i)$ y $x_j = g^{a_j} h^{b_j}$. Como la función $f$ ha de tener un comportamiento pseudoaleatorio, Pollard sugiere dividir los elementos de $G$ en tres grupos distintos $T_1$, $T_2$ y $T_3$; y el uso de la siguiente función:

$$f(x) = \begin{cases} gx, & \text{si } x \in T_1 \\ x^2, & \text{si } x \in T_2 \\ hx, & \text{si } x \in T_3 \end{cases}$$

De este modo se puede definir una función $F$ para tripletas que retorna $(x_j, a_j, b_j) = F(x_i, a_i, b_i)$ con $x_j = f(x_i)$, de la siguiente manera,

$$F(x_i, a_i, b_i) = \begin{cases} (gx, a_i + 1, b_i), & \text{si } x \in T_1 \\ (x^2, 2a_i, 2b_i), & \text{si } x \in T_2 \\ (hx, a_i, b_i + 1), & \text{si } x \in T_3 \end{cases}$$

siendo las operaciones sobre $a_i$ y $b_i$ módulo $p$.

Comenzando desde una tripleta cualquiera $(x_0, a_0, b_0)$, se puede construir la secuencia $\{(x_i, a_i, b_i)\}_{i\geq 0}$ donde $(x_i, a_i, b_i) = F(x_{i-1}, a_{i-1}, b_{i-1})$ para $i \geq 1$. Dado que el grupo $G$ es finito, existe un índice $t$ para el cual $x_t = x_{t-s}$ para algún $s \geq 0$. Entonces $x_i = x_{i-s}$ para todo $i \geq t$ con lo que la secuencia entre en un ciclo.

Partiendo de la paradoja del aniversario se demuestra que el camino generado antes de que se produzca la primera colisión (valor del índice $t$) tendrá una longitud esperada de $\sqrt{\frac{\pi p}{2}}$. El algoritmo Rho de Pollard encuentra una de estas colisiones como dos tripletas distintas $(x_i, a_i, b_i)$, $(x_j, a_j, b_j)$ con $x_i = x_j$, y resuelve el logaritmo discreto aplicando la fórmula 3.1.

## 3.1. Algoritmos para la detección de ciclos

Tal y como se ha visto en el apartado anterior, la resolución del problema mediante el algoritmo *Rho de Pollard* implica la detección de un ciclo en en una secuencia hallando así una colisión ($x_i = x_j$, $(x_i, a_i, b_i) \neq (x_j, a_j, b_j)$ ) lo antes posible. Para ello existen algoritmos para encontrar un ciclo dentro de una secuencia dada. Los algoritmos de *Floyd* y *Brent* se pueden usar para este fin.

### 3.1.1. Algoritmo de Floyd

Uno de estos algoritmos para la búsqueda de ciclos es el algoritmo de Floyd o algoritmo de "la tortuga y la liebre". Este algoritmo utiliza dos tripletas que avanzan a lo largo de una misma secuencia que comienza en una tripleta cualquiera $(x_0, a_0, b_0)$. La primera de ellas, denominada tortuga, avanza un paso en cada iteración, mientras que la otra tripleta, denominada liebre, avanza dos pasos por iteración. Una vez las dos tripletas hayan entrado en un ciclo, habrá un momento en el que la liebre alcance a la tortuga hallándose una colisión.

**Algoritmo 1** Algoritmo de Floyd

**Entrada:** Una tripleta inicial $(x_0, a_0, b_0)$
**Salida:** Dos tripletas distintas $(x_T, a_T, b_T)$, $(x_L, a_L, b_L)$ con $x_T = x_L$

1: $(x_T, a_T, b_T) := (x_0, a_0, b_0)$;
2: $(x_L, a_L, b_L) := F(x_T, a_T, b_T)$;
3: **mientras** $x_L \neq x_T$ **hacer**
4: $\quad (x_T, a_T, b_T) := F(x_T, a_T, b_T)$;
5: $\quad (x_L, a_L, b_L) := F(F(x_L, a_L, b_L))$;
6: **fin mientras**
7: **devolver** $(x_T, a_T, b_T)$, $(x_L, a_L, b_L)$

Este algoritmo solamente necesita memoria para almacenar dos tripletas y el número de operaciones que hay que realizar sobre el grupo para encontrar una colisión es, aproximadamente, $3\sqrt{p}$.

### 3.1.2. Algoritmo de Brent

El algoritmo de Brent para la detección de ciclos permite encontrar colisiones mediante una técnica distinta también basada en el uso de dos tripletas llamadas tortuga y liebre. En este algoritmo, la liebre avanza un paso en cada iteración mientras que la tortuga permanece quieta. La tortuga toma el valor de la liebre una vez esta última ha realizado $2^i$ saltos desde la última actualización de la tortuga. El índice $i$ toma un valor inicial de 1 y es incrementado en una unidad cada vez que la tortuga es actualizada, por tanto en las primera iteraciones del bucle, la liebre daría dos pasos antes de actualizar la tortuga, después daría cuatro pasos, antes de actualizar, ocho pasos, y así sucesivamente.

**Algoritmo 2** Algoritmo de Brent

**Entrada:** Una tripleta inicial $(x_0, a_0, b_0)$
**Salida:** Dos tripletas distintas $(x_T, a_T, b_T)$, $(x_L, a_L, b_L)$ con $x_T = x_L$

1: $(x_T, a_T, b_T) := (x_0, a_0, b_0)$;
2: $(x_L, a_L, b_L) := F(x_T, a_T, b_T)$;
3: $Longitud := 1$;
4: $Pasos := 1$;
5: **mientras** $x_L \neq x_T$ **hacer**
6: **si** $Pasos = Longitud$ **entonces**
7: $(x_T, a_T, b_T) := (x_L, a_L, b_L)$;
8: $Longitud := 2 * Longitud$;
9: $Pasos := 0$;
10: **fin si**
11: $(x_L, a_L, b_L) := F(x_L, a_L, b_L)$;
12: $Pasos := Pasos + 1$;
13: **fin mientras**
14: **devolver** $(x_T, a_T, b_T)$, $(x_L, a_L, b_L)$

Igual que el algoritmo de Floyd, el algoritmo de Brent tiene un coste de memoria insignificante, pero en [1] se afirma que su utilización incrementa la velocidad del algoritmo Rho de Pollard alrededor de un 24 % .

## 3.2. Parelelización

### 3.2.1. Paralelización Trivial

Cuando se dispone de varios procesadores, el algoritmo Rho de Pollard puede paralelizarse de forma trivial haciendo que cada procesador ejecute el algoritmo de forma independiente partiendo de una tripleta inicial distinta. El algoritmo paralelizado se detiene en el momento en que alguno de los procesadores halla una colisión.

Si se dispone de $M$ procesadores y se buscan colisiones mediante el algoritmo de Floyd o el algoritmo de Brent, lo cual permite un uso negligible de memoria, el número esperado de operaciones sobre el grupo realizadas por cada procesadores en el momento de encontrar la primera colisión es de $3\sqrt{\frac{p}{M}}$ con lo que la propuesta proporciona un *speedup* que es solamente un factor de $\sqrt{M}$. Esto

resulta muy poco eficiente. Por ejemplo, si se quisiera reducir a la cuarta parte el tiempo de cálculo, se habrían de emplear 16 procesadores.

### 3.2.2. Búsqueda paralela de colisiones mediante tripletas distinguidas

En [7] se propone una versión paralela de búsqueda de colisiones que permite paralelizar el algoritmo Rho de Pollard con un *speedup* proporcional a $M$ cuando se dipone de $M$ procesadores y de una zona de memoria para almacenar tripletas.

En esta propuesta, cada procesador escoge una tripleta al azar $(x_0, a_0, b_0)$ desde la cual construye un camino de tripletas $(x_i, a_i, b_i) = F = (x_{i-1}, a_{i-1}, b_{i-1}), i > 0$, hasta encontrar un valor $x_d$ que satisface una determinada condición de fácil comprobación. Esta tripleta distinguida,$(x_d, a_d, b_d)$ es enviada a un procesador encargado de recoger y almacenar las tripletas distinguidas que le irán mandando los otros procesadores. El procesador que halló la tripleta distinguida, después de mandarla, generará una nueva tripleta inicial al azar y repetirá el mismo proceso. El algoritmo termina cuando el procesador encargado de recoger las tripletas distinguidas recibe una nueva tripleta $(x_d, a_d, b_d)$ cuyo valor $x_d$ coincide con el de otra tripleta distinta almacenada previamente en la lista. En este momento se produce una colisión y el algoritmo termina.

La propiedad que distingue a algunos elementos del grupo $G$ ha de escogerse de manera que la proporción de elementos que la satisfacen sea $\theta$. Dado $\theta$, el número de operaciones sobre el grupo realizadas por cada procesador antes de hallarse la primera colisión, es decir, el coste temporal es:

$$\frac{1}{M}\sqrt{\frac{\pi p}{2}} + \frac{1}{\theta} \tag{3.2}$$

Puede ocurrir que la secuencia calculada por alguno de los procesadores entre en un ciclo sin puntos distinguidos. Una forma de recuperarse de esta situación consiste en establecer una longitud de camino máxima. En [7] se recomienda un valor de $20/\theta$, valor que, en el momento de ser alcanzado por algún camino, es abandonado para iniciar el cálculo de otro camino desde una tripleta diferente generada al azar.

La fórmula 3.2 supone que hay suficiente memoria para almacenar todas las tripletas enviadas al procesador que las almacena. Como se dispone de $M$ procesadores trabajando en paralelo y cada uno de ellos envía una proporción de $\theta$ tripletas a la lista, el número de tripletas que serán almacenadas es:

$$\theta\sqrt{\frac{\pi p}{2}} + M \tag{3.3}$$

El parámetro $\theta$ ajusta el compromiso entre los costes temporal y espacial. Un valor grande de $\theta$ reduce el coste temporal a expensas de aumentar el almacenaje y el número de tripletas enviadas lo que, por otra parte, podría llegar a causar un cuello de botella.

Cuando la memoria disponible es limitada, existen dos opciones. La primera consiste en tomar el valor más grande de $\theta$ posible, de manera que el almacenaje previsto necesario y calculado por 3.3, corresponda con la memoria disponible. La segunda opción, sugerida en [7] es la de tomar un $\theta$ más grande y añadir tripletas a la lista hasta que la memoria se agote. A partir de este momento, cada vez que se deba almacenar una tripleta nueva, se eliminará alguna de las almacenadas previamente.

Esta propuesta proporciona un *speedup* que al aumentar linealmente con el número de procesadores resulta eficiente. Por otro lado, el almacenaje de tripletas distinguidas obliga a disponer de un espacio de memoria cuyo mayor tamaño repercute en el coste temporal del algoritmo. A mayor tamaño, menor es el tiempo de ejecución.

### 3.2.3. Paralelización con requisitos de memoria negligibles

En este proyecto se ha implementado una nueva forma de paralelizar el algoritmo Rho de Pollard que proporciona un *speedup* proporcional al número de procesadores disponibles y que requiere del uso de muy poca memoria. El resultado de este trabajo ha sido publicado en [6].

En esta propuesta cada uno de los procesadores ejecuta el algoritmo de Pollard buscando la colisión de dos tripletas mediante el algoritmo de Brent tomando una tripleta al azar como punto de partida de su secuencia. Sin añadir nada más, esto sería una simple paralelización trivial con la que se obtendría un *speedup* proporcional a $\sqrt{M}$.

Para disminuir el tiempo de cálculo y conseguir una aceleración proporcional a $M$ se modifica la paralelización trivial haciendo que en cada operación, cada procesador compruebe si su liebre ha alcanzado, no sólo a su tortuga, sino también a alguna del resto de procesadores.

Así, se precisa de una estructura de memoria accesible por todos los procesadores dentro de la misma máquina que, además, implemente una estructura de datos de acceso rápido como puede ser un *hash*.

**Algoritmo 3** Paralelización del algoritmo Rho de Pollard con coste de memoria negligible

**Entrada:** Un grupo $G$ de orden $p$ y dos elementos $g, h \in G$
**Salida:** El logaritmo discreto $m = \log_g h$

1: T:=tabla de tripletas de $i$ posiciones.
2: Cada procesador $i$ hace:
3: Escoger $a_L, b_L \in [0, p-1]$ aleatorios;
4: $x_L := g^{a_L} \cdot h^{b_L}$;
5: T[$i$]:= $(x_L, a_L, b_L)$;
6: $(x_L, a_L, b_L) := F(x_L, a_L, b_L)$;
7: *Longitud* := 1;
8: *Pasos* := 1;
9: **mientras** No haya en la tabla una tripleta $(x_T, a_T, b_T) \neq (x_L, a_L, b_L)$ con $x_T = x_L$ **hacer**
10: **si** *Pasos* = *Longitud* **entonces**
11: T[$i$]:=$(x_L, a_L, b_L)$;
12: *Longitud* := 2 $*$ *Longitud*;
13: *Pasos* := *0*;
14: **fin si**
15: $(x_L, a_L, b_L) := F(x_L, a_L, b_L)$;
16: *Pasos* := *Pasos* + 1;
17: **fin mientras**
18: Recuperar la tripleta $(x_T, a_T, b_T)$ con $x_T = x_L$ de memoria.
19: Ordenar a todos los procesadores que se detengan;
20: $m$:=$(a_T - a_L)(b_L - b_T)^{-1} \pmod p$;
21: **devolver** $m$

# Capítulo 4

# Diseño

En este proyecto se ha implementado el algoritmo detallado en la sección anterior en un clúster que tiene el protocolo de paso por mensajes MPI, de modo que el algoritmo pasa de utilizar memoria compartida entre procesos y utilizar memoria distribuida entre los diferentes ordenadores del clúster.

En esta propuesta, cada proceso tiene la misión de, no sólo almacenar su tortuga, sino que también ha de comunicar al resto de procesadores cada nueva tortuga a la vez que recibe y almacena las tortugas individuales del resto de procesos. Así una versión del algoritmo utilizanzo diferentes ordenadores conectados y un protocolo de paso por mensajes quedaría tal y como puede verse en el algoritmo 4

Como puede apreciarse, en el algoritmo 3 no se incluyen detalles de la comunicación ya que está escrito en pseudocódigo de alto nivel y está pensado para procesos dentro de una misma máquina. En el algoritmo 4 se detallan las comunicaciones (que corresponden a las líneas de envío y recepción de mensajes), ya que se trabaja en un entorno de comunicación de paso por mensajes y memoria distribuida y, en este sentido, las comunicaciones cobran importancia.

## 4.1. Estructuras de datos

Para almacenar las tortugas se ha utilizado un *hash* doble enlazado, en el que se guardan las tripletas (tortuga,a,b), combinado con una tabla donde se guardan apuntadores a las tortugas de cada procesador. Esta segunda tabla es útil para acceder a las tortugas de cada procesador mediante acceso directo por el identificador del procesador, esto mejora el rendimiento a la hora de sustituir las tripletas de los diferentes procesadores, ya que mejora la gestión de la memoria y permite relacionar la tortuga a borrar con el procesador al que pertenecía.

**Algoritmo 4** Pseudocódigo del algoritmo implementado en este proyecto con comunicación de procesos por paso de mensajes

**Entrada:** Un grupo $G$ de orden $p$ y dos elementos $g, h \in G$
**Salida:** El logaritmo discreto $m = \log_g h$

```
1: Inicializar estructuras de memoria;
2: Cada procesador i hace:
3: Escoger a_L, b_L ∈ [0, p − 1] aleatorios;
4: x_L := g^{a_L} · h^{b_L};
5: Almacenar tortuga inicial propia;
6: Enviar tortuga inicial propia;
7: Recibir y almacenar tortugas iniciales del resto de procesos;
8: (x_L, a_L, b_L) := F(x_L, a_L, b_L);
9: Longitud := 1;
10: Pasos := 1;
11: mientras No haya en la tabla una tripleta (x_T, a_T, b_T) ≠ (x_L, a_L, b_L) con
    x_T = x_L hacer
12:   si Pasos = Longitud entonces
13:     Sustituir tortuga propia por (x_L, a_L, b_L);
14:     Enviar tortuga (x_L, a_L, b_L) al resto de procesos;
15:     Recibir tortugas del resto de procesos y sustituirlas;
16:     Longitud := 2 ∗ Longitud;
17:     Pasos := 0;
18:   fin si
19:   (x_L, a_L, b_L) := F(x_L, a_L, b_L);
20:   Pasos := Pasos + 1;
21: fin mientras
22: Recuperar la tripleta (x_T, a_T, b_T) con x_T = x_L de memoria.
23: Ordenar a todos los procesadores que se detengan;
24: m:=(a_T − a_L)(b_L − b_T)^{−1} (mód p);
25: devolver m
```

## 4.2. Clases

### 4.2.1. Clase *node*

La clase node es una clase sin parte privada para permitir el acceso directo a los campos sin utilizar funciones. Representa un nodo, un elemento compositivo de la estructura principal del almacenamiento de datos del programa. Guarda la información de una tripleta de (tortuga,a,b) pero, además de los valores *tortuga*, $a$ y $b$; la clase nodo tiene dos apuntadores a los nodos anterior y siguiente para poder programar el *hash* doble enlazado con comodidad como puede observarse en la figura (4.1).

### 4.2.2. Clase *hash_pollard*

La clase *hash_pollard* es la clase que representa el doble *hash* enlazado. Contiene un vector de *node* y el tamaño en número de elementos de ese vector. Esta clase ha sido inicialmente implementada para la paralelización del algoritmo Rho de Pollard para u na máquina con procesador multinúcleo, aunque para la

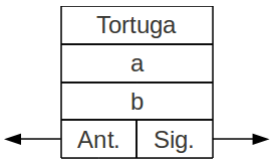


Figura 4.1: Representación gráfica de la clase nodo.

realización del programa paralelizado con MPI se ha modificado prácticamente en su totalidad para poder hacer una mejor gestión de la memoria y el acceso directo a la estructura. Define operaciones de inicialización de la estructura; la función de *hash*, para un acceso directo rápido y eficiente; funciones de inserción, consulta y borrado de nodos y apuntadores. En la figura 4.2 que representa el diseño general de las relaciones entre las diferentes estructuras de datos, se corresponde con la tabla coloreada en gris claro.

### 4.2.3. Clase *hash_processors*

La clase *hash_processors* consiste en una lista de apuntadores a objetos de la clase *node*. En esta lista existen tantos elementos como número de total de procesadores haya trabajando en el problema en concreto, y cada uno de esos apuntadores contiene la dirección de memoria del nodo con el que está trabajando cada procesador, es decir, su tripleta de (tortuga,a,b) dentro de la estructura principal de la clase *hash_pollard*.

Esta clase es necesaria para poder acceder al nodo directamente y sin necesidad de buscar el valor dentro del hash. Este hecho permite acceder de manera directa a la hora de modificar en la lista principal cada vez que un procesador cambia su tortuga. En la figura 4.2 esta estructura se corresponde con la tabla de apuntadores representada en color gris oscuro.

Como puede observarse, gracias a esta disposición de las estructuras de datos en memoria (figura 4.2), se puede acceder a los valores de la tabla principal, tanto por el valor de la *tortuga*, como por el número de procesador que está trabajando con un valor determinado y de ambas formas mediante acceso directo. Así, si en un momento determinado del programa, se requiere cambiar el valor de la tortuga del procesador $x$ podemos borrar directamente de la tabla y con acceso directo el nodo asociado a ese procesador, y añadir el nuevo valor en la tabla *hash* para después asociarlo con la posición $x$ de la tabla *hash_processors*.

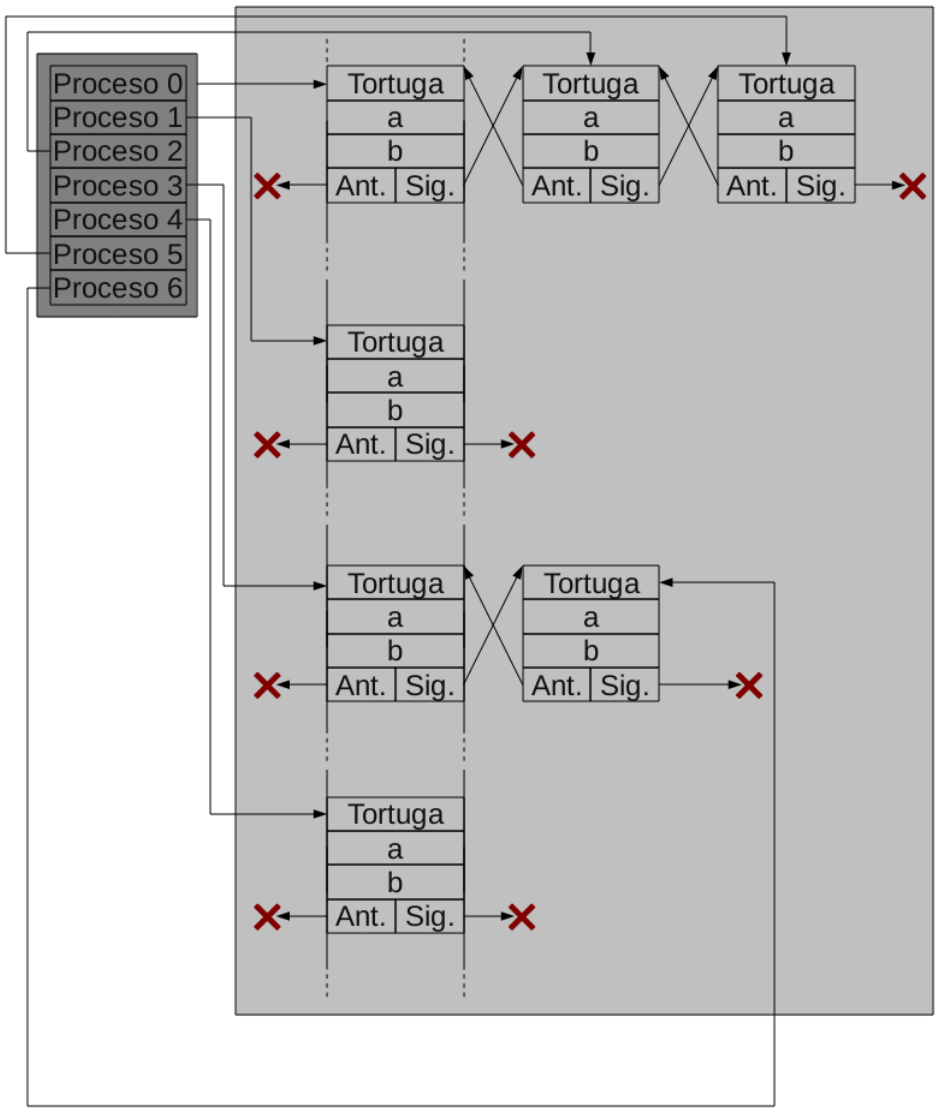


Figura 4.2: Representación gráfica de un estado posible de la memoria durante una ejecución del programa con 7 procesos dedicados a la resolución del problema.

## 4.3. Protocolo de paso por mensajes MPI

Debido a que los algoritmos relacionados con algunos casos de criptoanálisis comportan en sí una carga computacional muy elevada, se hace inadmisible el tiempo que dedica una sola máquina a la ejecución de éstos. Así pues, se pone de manifiesto la necesidad soluciones que impliquen la ejecución cordinada de varias máquinas para resolver un mismo problema, esto se llama la computación distribuida.

Para la implementación de la resolución del problema del logaritmo discreto, en este proyecto, se ha utilizado un sistema de computación paralela mediante procesos intercomunicados por un protocolo de paso de mensajes. No se trata de un lenguaje de programación específico, sino más bien de una librería que contiene un reducido número de funciones para conseguir paralelismo mediante el paso de mensajes.

Los protocolos de paso de mensajes son un método muy habitual y potente de expresar paralelismo; y la facilidad y la posibilidad de crear programas paralelos extremadamente eficientes, han convertido a los sistemas de paso de mensajes en la manera más ampliamente extendida de programar aplicaciones para multicomputadores.

El sistema de paso por mensajes MPI (Message-Passing Interface) es un conjunto de funciones programadas para los lenguajes de programación C, C++ y Fortran que permite desarrollar programas paralelos mediante el paso de mensajes de una manera eficiente en estos tres lenguajes. Para este proyecto se ha elegido la variante de C++ que aúna la potencia del sistema de paso por mensajes con la facilidad y flexibilidad que permiten las clases de programación de un lenguaje orientado a objetos.

## 4.4. Librería de números grandes NTL

Conforme aumentan las capacidades de cálculo de las computadoras, se va haciendo necesario incrementar en número de bits de las claves criptográficas, hasta tal punto que los lenguajes de programación no implementan estructuras que puedan abarcar números enteros de según qué cantidades de bits. Así aparecen las librerías de enteros grandes que proporcionan estructuras donde se puede dar cabida a los numeros enteros que, por su tamaño, no son soportados internamente por los lenguajes de programación.

La librería NTL es una librería escrita en C++, portable y de gran eficiencia que proporciona estructuras de datos y funciones para manipular longitudes arbitrarias de enteros, vectores, matrices y poliniomios; tanto sobre números enteros, como sobre cuerpos finitos.

En este trabajo, se ha utilizado la clase ZZ de la librería NTL, que permite trabajar con enteros de tamaño arbitrario sin definir un máximo más allá de las limitaciones de memoria de cada ordenador. Dicha clase proporciona un

conjunto de funciones de aritmética modular que han resultado realmente útiles para la realización de este proyecto, así como transformaciones de los enteros de la clase ZZ a bytes físicos de memoria y viceversa, lo que ha facilitado el envío de datos entre los diferentes nodos de trabajo del clúster.

## 4.5. Otras cuestiones de implementación

### 4.5.1. Tamaño de la tabla de tripletas

Como la tabla de tripletas es de tipo *hash*, la tabla tiene que tener más tamaño que el número de los elementos que vaya a albergar para evitar acumulaciones y ristras de objetos seguidos en la tabla. Para la implementación de este proyecto se ha elegido un número que puede ser exagerado, pero que por el tamaño de los números y el número de procesadores no produce un gasto excesivo de memoria. En concreto el número de objetos de la clase *node* que puede albergar la tabla principal, es el número primo siguiente al resultado de multiplicar el número de procesadores ejecutando el problema en concreto por 16. La elección del número primo es debido a que la dispersión de la función *hash* es mejor si el tamaño de la tabla es un número primo. Esto se ha hecho, y es una práctica común en el desarrollo de *hashes* para evitar la tendencia de que los *hash* de enteros grandes tengan divisores comunes con el tamaño de la tabla, lo que provocaría colisiones tras el cálculo de la función módulo.

En el caso de trabajar con números más grandes (mayor número de bits) y/o de que las pruebas se realizaran sobre un *grid* con un gran número de computadores, habría que bajar el factor 16, para ser más consecuentes con la utilización de la memoria.

### 4.5.2. Representación de las tripletas en memoria

Con el fin de permitir una mayor flexibilidad a la hora de enviar las tripletas, y aprovechando las funciones ya implementadas en la librería NTL para números grandes, en este proyecto se convierten las tripletas a bytes. Esto se hace utilizando las funciones de la librería NTL *BytesFromZZ* y *ZZFromBytes* que permiten expresar un entero *grande* del tipo *ZZ* como una cadena de bytes. Dado que es conocido para cada problema en particular, el número de bits del número mayor con el que se trabajará, no conlleva una dificultad especial el hecho de trabajar reservando memoria con las funciones primitivas del lenguaje C del tipo *malloc*.

Toda esta transformación en la que se reserva memoria continua para tres enteros del tipo ZZ y se escriben uno detrás de otro en esa memoria con la función *BytesFromZZ*, es necesaria debido a que al protocolo MPI se le ha de especificar el tipo de datos que se va a enviar, estando éstos predefinidos a los tipos estándar de C. La única posibilidad era enviar estos números como una ristra de bytes, algo que sí se define como un tipo, según la especificación MPI [3].

# Capítulo 5

# Experimentación

## 5.1. Experimentos realizados

El programa se ha ejecutado en un clúster con 8 nodos tetraprocesador, en los cuales los procesadores son Intel Quad Core con una velocidad de reloj de 2,66 GHz en cada procesador.

Se han realizado pruebas experimentales del algoritmo con 20 problemas diferentes del logaritmo discreto escogidos al azar con números enteros *grandes* de diferentes longitudes, en concreto 52,...,64 bits; utilizando también diferente número de procesadores en todos y cada uno de los problemas. Concretamente, para cada problema se han realizado pruebas experimentales con 1, 2, 4, 6, 8, 16, 24 y 32 procesadores.

Se pueden distinguir dos grupos de problemas según se involucren uno o más procesadores por máquina. Así, en los problemas ejecutados con un número de procesos menor o igual al número de máquinas disponibles, se dedica un sólo procesador de cada máquina al cálculo del problema.

En cuanto a los problemas ejecutados sobre un número de procesadores mayor a 8, se reparte equitativamente el número de procesadores dedicado al problema dentro de cada máquina. En los problemas ejecutados sobre 16 procesadores, se utilizaban 2 procesadores de cada máquina; en los ejecutados sobre 24 procesadores, se utilizaban 3 por máquina; y en los de 32, se usaban todos los recursos disponibles, es decir, los 4 procesadores de cada máquina.

Posteriormente se han realizado más pruebas con números enteros de mayor tamaño, en concreto 70 y 74 bits. Dado que el tiempo que se dedica es mucho mayor conforme aumenta el número de bits, de estas pruebas, sólo se han ejecutado 10 problemas diferentes para 8, 16, 24 y 32 procesadores y siguiendo la misma compensación de procesadores por máquina que en las pruebas anteriores; quedando así pruebas a realizar en un futuro, con el fin de homogeneizar los resultados.

Al término de la recolección de todos los resultados, debidamente registrados en un documento de hoja de cálculo, la totalidad de los tiempos es de 2542569 segundos de trabajo, lo que hace un total de 706, 27 horas, es decir, 29′42 días de ejecución en los 8 nodos disponibles. Y este periodo de tiempo, casi un mes, se refiere solamente a las pruebas finales realizadas, sin tener en cuenta las pruebas iniciales, las que no están registradas ni las que fallaron antes de tener el programa debidamente programado.

## 5.2. Resultados

A continuación se muestran las tablas con los resultados temporales medios obtenidos, y el *speedup* calculado para todas las ejecuciones realizadas del algoritmo.

| | Número de bits | | | | | |
|---|---|---|---|---|---|---|
| Procesadores | 52 | 56 | 60 | 64 | 70 | 74 |
| 1 | 241 | 1260 | 1844 | 16921 | | |
| 2 | 130 | 829 | 1015 | 8160 | | |
| 4 | 58 | 337 | 689 | 4543 | | |
| 6 | 45 | 265 | 582 | 2799 | | |
| 8 | 40 | 258 | 469 | 2092 | 23892 | 97271 |
| 16 | 8 | 96 | 59 | 690 | 8764 | 43374 |
| 24 | 2 | 76 | 60 | 527 | 4758 | 25901 |
| 32 | 2 | 64 | 63 | 613 | 4625 | 24088 |

Cuadro 5.1: Tabla de tiempos (en segundos)

| | Número de bits | | | | | |
|---|---|---|---|---|---|---|
| Procesadores | 52 | 56 | 60 | 64 | 70 | 74 |
| 1 | 1 | 1 | 1 | 1 | | |
| 2 | 1,85 | 1,52 | 1,82 | 2,07 | | |
| 4 | 4,16 | 3,74 | 2,68 | 3,72 | | |
| 6 | 5,36 | 4,75 | 3,17 | 6,05 | | |
| 8 | 6,03 | 4,88 | 3,93 | 8,09 | 8 | 8 |
| 16 | 30,13 | 13,13 | 31,25 | 24,52 | 21,84 | 17,92 |
| 24 | 120,5 | 16,58 | 30,73 | 32,11 | 40,16 | 30,08 |
| 32 | 120,5 | 19,69 | 29,27 | 27,6 | 41,36 | 32,32 |

Cuadro 5.2: Tabla de *speedup*

Nota: Los valores de *speedup* para los problemas ejecutados con números de 70 y 74 bits, corresponden a los valores medios de 10 problemas diferentes ejecutados con 8, 16, 24 y 32 procesadores cada uno. Debido a la gran cantidad de tiempo que se ha de dedicar a cada uno de estos problemas, para este cálculo se ha tomado como referencia el valor temporal medio de las 10 ejecuciones de los 10 problemas propuestos ejecutados sobre 8 procesadores.

## 5.3. Análisis de resultados

Debido a que las decisiones del algoritmo se hacen mediante una función heurística, los resultados no dejan de tener un caracter aleatorio, si bien es verdad que, aunque siempre exista una varianza elevada, el *speedup* se acercaba al número de procesadores conforme aumentaba el número de repeticiones del algoritmo con diferentes problemas a tratar, y tendía a estabilizarse.

En cuanto a los resultados obtenidos, se puede decir que son satisfactorios en cuanto a la paralelización del algoritmo se refiere, ya que los valores de *speedup* obtenidos, en general, son bastante cercanos al número de procesadores dedicados al problema, sobre todo conforme se iba aumentando el número de bits y el número de procesadores trabajando en ello. Se observan peores resultados, en general, con el máximo número de procesadores disponible, cuatro por máquina, que con 24 procesadores involucrados (3 por máquina). Esto puede ser debido a que, en los problemas con el máximo número de procesadores involucrados, los procesos propios del sistema operativo precisan de unos ciclos de procesador que "roban" a alguno de los procesadores calculando el problema de turno; mientras que en los problemas que no involucran a todos los procesadores, el sistema operativo, bien programado y compilado teniendo en cuenta el número de procesadores presente en la máquina, utiliza, para cualquier proceso, un procesador ocioso si es que este está disponible.

Además otro factor que puede haber tenido algo que ver en este peor resultado relativo de 32 procesos frente a 24, puede tener su origen en que la cantidad de mensajes que se producen según el programa es cuadrática en cuanto al número de procesadores involucrados como se aprecia en la fórmula 7.1. Esto puede provocar sobrecarga de la red o el llenado de los búferes que MPI dispone para las conexiones y el paso de mensajes asíncrono.

Como se podía prever, el tiempo para enteros de tamaño *grande* se dispara incluso cuando se ejecuta el programa con tantos procesos como procesadores hay disponibles.

# Capítulo 6

# Conclusiones

## 6.1. De los resultados

Vistos los resultados y una vez hecho un análisis de los mismos se extraen una serie de conclusiones que se listan a continuación:

- El algoritmo paralelizado con MPI mantiene hasta donde se ha podido llegar una progresión más o menos lineal tanto en los tiempos como en los procesadores.
- Para obtener unos resultados más homogéneos, hubiera sido deseable ejecutar más conjuntos de pruebas, pero el cómputo de las resoluciones de problemas de los enteros con mayor número de bits hacia imposible la ejecución en pocos procesadores en un tiempo razonable.
- El problema del logaritmo discreto, con el tamaño de lo número que se usa en producción en la actualidad, es inabordable para un clúster pequeño siguiendo el algoritmo Rho de Pollard.
- La cantidad de tiempo necesario para abordar un problema en concreto mediante el algoritmo Rho de Pollard paralelizado con MPI, es un valor con una varianza muy alta, debido a que la selección de las siguientes tripletas de la secuencia se realiza mediante una función heurística.
- El incremento de las realizaciones de los problemas estabiliza la media de tiempos, es decir, aunque la función sea heurística, el resultado temporal de las realizaciones tiende a alcanzar un valor medio conforme se aumentan las repeticiones del algoritmo con diferentes problemas.
- El *speedup* de los resultados mejora al algoritmo original en muchos de los casos.
- El *speedup* y el tiempo de los valores para 32 procesadores es peor, relativamente, al de 24 procesadores. Esto puede querer decir que se ha llegado al punto de inflexión en el que el elvado número de comunicaciones afectan al rendimiento total del sistema, ya sea por la cantidad de mensajes en la red y el encapsulamiento que conllevan, o por el llenado de los búferes de envío y recepción asíncrona de MPI.

- Para abordar problemas mayores, en cuanto a número de bits se refiere, no basta con un clúster de 8 máquinas tetraprocesador, llega un momento en que se produce un punto de inflexión y hay que pensar en algo más grande, un sistema en *grid* podría ser una de las soluciones.

## 6.2. Del trabajo

- MPI es un entorno de trabajo y programación adaptable a las necesidades de un programa paralelo.
- La comunicación con MPI para trabajos en paralelos es sencilla de programar con comunicaciones punto a punto, ya sean síncronas o asíncronas, así como las comunicaciones uno a todos o *broadcast*.
- En términos de red, el hecho de tener una red y un switch dedicados al trabajo y aislados del resto de la red, ha podido influir de manera positiva en los resultados temporales de la aplicaciones; ya que, la latencia habrá sido más baja, y la red estaba exclusivamente dedicada al trabajo.
- Problemas con realizaciones temporales tan largas, precisan de una dedicación total de máquinas de alto rendimiento las 24 horas del día, y en investigación, en estos términos, cualquier error de procedimiento o de cálculo puede conllevar pérdidas masivas de tiempo en caso de no darse cuenta del error previamente a la ejecución.
- Se ha alcanzado el límite de las posibilidades del clúster en cuanto a requisitos temporales se refiere, haciendo varias ejecuciones del problema de 74 bits.

# Capítulo 7

# Líneas abiertas y trabajos futuros

## 7.1. Necesidad de un servidor

Conforme se aumenta el número de procesadores trabajando en un mismo problema, es de perogrullo que incrementa el número de comunicaciones entre los procesadores. Esto puede provocar un problema si existen muchos pequeños paquetes de datos que enviar mediante la red, ya que el encapsulamiento que conllevan los distintos protocolos de red puede ser más importante en tamaño que el dato en sí.

No es complicado calcular la cantidad de mensajes que circulan por la red. Teniendo en cuenta que, según la implementación actual, cada uno de los procesadores se intercambia un mensaje con cada uno del resto de los procesadores involucrados en el trabajo, en un caso genérico con $n$ procesadores trabajando, cada uno de ellos envía $n-1$ mensajes en cada ciclo del algoritmo. Así pues, el número de mensajes totales por ciclo de cómputo es:

$$n * (n-1) = n^2 - n \tag{7.1}$$

Esto es perfectamente asumible para un número de procesos como con el que en este trabajo se ha experimentado, habida cuenta de que trabajos en los que implican a todos los procesadores disponibles (32) y aplicando la fórmula 7.1, el número de mensajes que se intercambian los procesadores alcanza la nada desdeñable cantidad de 993 mensajes por ciclo de cómputo.

Podría hacerse necesaria, pues, la existencia de un servidor dedicado a sincronizar y centralizar las comunicaciones, con el fin de amortiguar el problema del encapsulamiento y rebajar el número de paquetes circulando por la red.

El proceso servidor, sería el proceso al que todos enviaran sus nuevas tripletas, y del que todos recibieran la lista completa y actualizada de tripletas, con lo

que se conseguiría reducir el número de mensajes por cada procesador y ciclo de ejecución. Así la cantidad de mensajes circulando por la red en cada ciclo de ejecución sería: uno con la tripleta a sustituir, y otro con la lista completa y actualizada de tripletas; para cada uno de los procesadores. Es decir $2n$. Hay que tener encuenta que en este caso $n$ es el número de procesadores dedicados a la resolución del problema en sí, mientras que el número total de procesadores trabajando en el programa sería de $n + 1$ ya que habría que añadir el proceso servidor.

## 7.2. Memoria compartida. Máquinas multinúcleo.

En cuanto a mejoras en la gestión de memoria se refiere, se podría empezar por evitar la redundancia de listas dentro de las máquinas multiprocesador. Tal y como está implementado el programa, cada procesador mantiene su propia lista de tripletas $(tortuga, a, b)$ actualizada. Esto es bueno de cara a la ejecución, porque si el entorno MPI es un conjunto de máquinas heterogéneas, el único cambio que tienes que hacer está en el fichero de máquinas MPI. En este fichero tienes que especificar cuantos núcleos tiene cada máquina, para que en cada una de ellas se lancen tantos procesos MPI como núcleos contiene su procesador correspondiente.

Sin embargo, esta manera de trabajar, tiene el problema de que pueden llegar a juntarse tantas listas idénticas como núcleos contiene el procesador, lo cual es una redundancia problemática conforme aumenta el número de bits y el número de procesadores trabajando en la resolución del problema.

Esta problemática se podría solucionar, en parte, guardando la lista en una zona de memoria compartida accesible por todos y cada uno de los procesos presentes en una misma máquina. La programación se complicaría bastante, debido a que habría que utilizar sistemas de intercomunicación de procesos, como los semáforos, para gestionar la memoria compartida; así como pensar en una manera de gestionar la lista de una manera conjunta entre los procesos de cada núcleo. En el caso de este proyecto, habiendo máquinas tetraprocesador involucradas, la memoria necesaria dentro de cada máquina hubiera sido, lógicamente, un cuarto de la memoria total empleada.

Idealmente, se podría pensar en un programa que detectara automáticamente el número de núcleos presentes en cada máquina y ejecutara tantos procesos hijos como número de procesadores. Además un sistema de funciones de hilo que gestionaran las actualizaciones de la lista y la comunicación con el exterior de manera rápida y eficaz gracias a la ligereza de los hilos de ejecución.

## 7.3. Salir al Exterior

El entorno MPI es un entorno cerrado y en el que todos los ordenadores tienen que estar interconectados en una red local. Además, tiene que ser un conjunto más o menos homogéneo en el que tiene que estar instalada la misma versión de

MPI en cada máquina. Esta serie de factores limita mucho las cosas a la hora de abordar un problema real.

Entonces, ¿qué hacer? Hay que pensar en algo más grande, en proyectos que superen los límites del clúster y se puedan extender a todos los ordenadores de la universidad, e incluso más allá. Un sistema *grid* en las salas de usuario de la universidad, centralizadas con un servidor y con comunicación basada en sockets; proporcionaría mucha más potencia al sistema, al poder utilizar mayor cantidad de máquinas. Es evidente que se trataría de otro proyecto distinto, en el que quedaría la esencia del problema matemático y el algoritmo de resolución, pero que cambiaría por completo el sistema de comunicación de las máquinas, el sistema de gestión de resultados, entre muchas otras cosas; y al que habría que añadir un sistema de recuperación de la estructuras en caso de caída del servidor mediante ficheros. Además la heterogeneidad de un sistema en *grid*, implica una programación adaptable en tiempo de ejecución o, al menos, en tiempo de instalación del programa en las máquinas, a la par que precisa de una tolerancia a fallos y caídas de los nodos de cómputo.

Dejar a un lado el entorno MPI y proceder con un sistema de funciones de comunicación propio conllevaría una dificultad mucho mayor y un conocimiento del lenguaje de programación mucho más avanzado.

# Bibliografía


[1] R. P. Brent, "An improved Monte Carlo factorization algorithm", BIT, vol. 20, pp. 176-184, 1980.

[2] T. ElGamal, "A public key criptosystem and a signature scheme based on discrete algorithm", IEEE Trans. Information Theory, vol. 31, pp. 469-472, 1985.

[3] MPI: Especificación MPI en MPI-forum.org. `http://www.mpi-forum.org/docs/`

[4] NTL: A library for doing Number Theory. `http://www.shoup.net`.

[5] J. M. Pollard, "Monte Carlo methods for index computation (mód $p$)", Mathematics of Computation, vol. 32 (143), pp. 918-924, 1978.

[6] F.Sebé, J.Pujolàs, T. Laírla, "Paralelización del algoritmo Rho de Pollard con requisitos de memoria negligibles", Actas de la XI Reunión Española sobre Criptología y Seguridad de la Información, pp. 79-83. ISBN 978-84-693-3304-4.

[7] P. van Oorschot, M. J. Wiener, "Parallel collision search with cryptanalytic applications", Journal of Cryptology, 12, pp. 1-28, 1999.